|  |
| --- |
| FAVV – AFSCA |
| favv.gif  C:\Users\emmnuy\AppData\Local\Microsoft\Windows\Temporary Internet Files\Content.MSO\8C168EFB.tmp |
| **[Technical Fiche : SpecFlow Automation Tests]** |

Document Information

|  |  |
| --- | --- |
| Information | Description |
| Document owner | Emmanuel Nuyttens |
| Creation date | 03/02/2020 |
| Last update date |  |
| Document version | 1.0 |

Document history

|  |  |  |
| --- | --- | --- |
| Version | Date | Modification |
| 1.0 | 03/02/2020 | Initial Created |
|  |  |  |
|  |  |  |
|  |  |  |

Document validation

|  |  |  |  |
| --- | --- | --- | --- |
| Nom | Role | Department | Date |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Document lecture references

|  |  |
| --- | --- |
| Title | Authors |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

Contents

[1. Preface 0](#_Toc31725137)

[2. High Level SpecFlow overview Steps 0](#_Toc31725138)

[3. Understanding SpecFlow Fundamentals 0](#_Toc31725139)

[3.1. Specflow Feature Files 101: 0](#_Toc31725140)

[3.2. Feature File Structure 0](#_Toc31725141)

[3.3. Domain Specific language : Gherkin 0](#_Toc31725142)

[ SpecFlow uses a domain specific language called Gherkin; https://github.com/cucumber/cucumber which is a business-readable type of language that let’s you describe your application’s context behavior without detailing how that behavior is implemented. 0](#_Toc31725143)

[ Gherkin provides different types of steps. So as explained before we have our scenario which contains a number of Steps. Each step will then map to a Gherkin step (Given;When;Then). 0](#_Toc31725144)

[4. Installation Overview 0](#_Toc31725145)

[5. The Use Case we will implement 0](#_Toc31725146)

[6. Implementing the test scenarios for Money 0](#_Toc31725147)

[6.1. Test-1 : Sum of 2 moneys produces correct result 0](#_Toc31725148)

[6.x. Test-X : xxx (Test-Template) 0](#_Toc31725149)

[7. Summary 0](#_Toc31725150)

[Annex A : Setup feature file 0](#_Toc31725151)

[ 0](#_Toc31725152)

1. Preface

Goal of this document is to give an introduction to SpecFlow. SpecFlow is mainly used to test our **domain model,** and more specifically to test if

the code that the developers build is what has been expected by the business. In ideal cases the functional SpecFlow scenario’s should have been

created by the PO[[1]](#footnote-1) or functional analyst in cooperation with the domain experts (in our case, the business).

Another goal of using Specflow in our projects is to **improve the communication between the business stakeholders, the product owner and the**

**development team**, this by writing tests that can be executed, and are at the same time **readable by the business.**

SpecFlow is based on **herkin language,** which is a kind of natural language, expressed in English (or any other language if you wish …).

So SpecFlow is all about **creating business readable tests, which are tests that run automatically to verify that a system is working as expected**, and

at the same time **document the system in a way that non-technical people can understand and contribute to**.

1. High Level SpecFlow overview Steps

|  |  |
| --- | --- |
|  | **Step1 : SpeFlow Feature File**  We start with a SpecFlow feature file. Here we write our high-level business language tests. Here we describe the different features and scenario’s that comprehend our application context workflow. |
|  | **Step2 : Map to C# Test Code Methods**  Next we will map the functional feature file to a number of test methods in C#. These can either be generated by the SpecFlow tool or hand code them. |
|  | **Step 3 : Test Automation Code**  Within the methods of the C# classes we will write our technical custom test automation code. In most cases we will create instances of our domain model (our parts of our domain model), execute some methods and check if we get the expected results as required by the SpecFlow scenario’s. |

1. Understanding SpecFlow Fundamentals
   1. Specflow Feature Files

* Understanding feature files.
  + Consist of a feature header.
  + A number of scenario’s that contain scenario steps:
    - “given”
    - “when”
    - “then”
  + Comments and tags in feature files.
  + Step definition code : mapping the functional to the technical.
  1. Feature File Structure

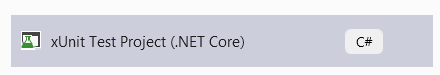
|  |  |
| --- | --- |
|  | * **Header :** contains the name and the description of the feature we want to create automated tests for. * **Scenario:** next each feature file contains a number of scenario’s which define the aspects of a feature, each scenario has a scenario name and contains a number of scenario steps. These steps describe the scenario in more detail (in a high-level, business oriented way). |

* 1. Domain Specific language : Gherkin

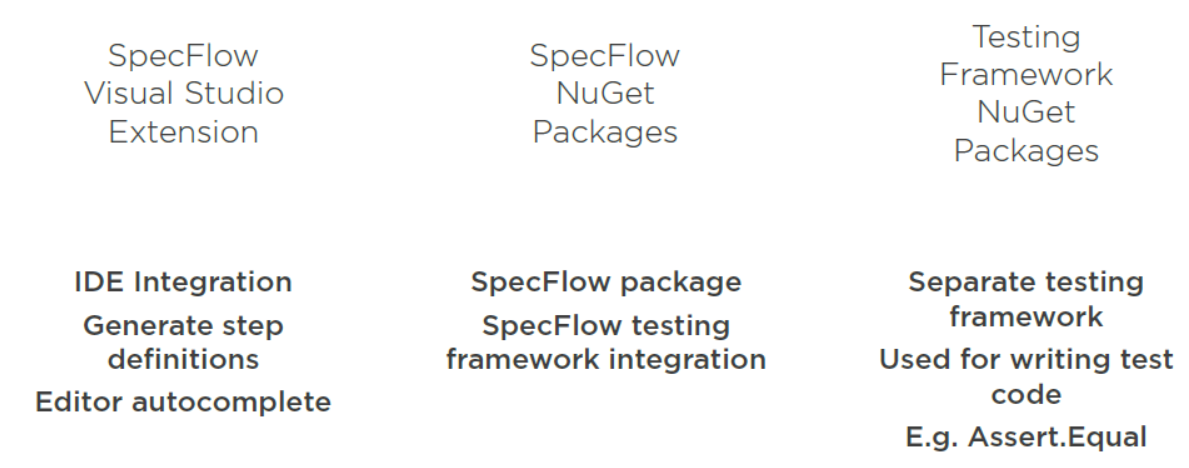
|  |  |
| --- | --- |
|  | * SpecFlow uses a domain specific language called Gherkin; <https://github.com/cucumber/cucumber> which is a business-readable type of language that let’s you describe your application’s context behavior without detailing how that behavior is implemented. * Gherkin provides different types of steps. So as explained before we have our scenario which contains a number of Steps. Each step will then map to a Gherkin step (Given;When;Then). |

1. Installation Overview

First we create a new regular xUnit test project in Visual Studio:



Next,There are 3 distinct things that we need to install:

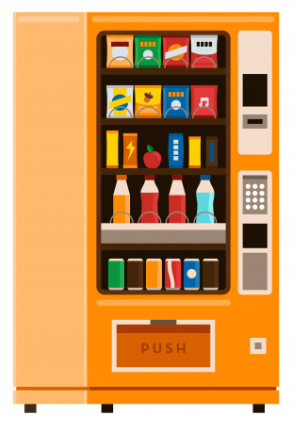


|  |
| --- |
| Install Specflow extension: |
|  |
| Install NuGet packages: |
|  |

1. The Use Case we will implement

As a demo the use case we will implement is the use case of the working of a simple vending machine (you know,

where you can get a snack or a drink and pay with coins …), something like this one:



Our domain model we have to test is quit simple, it only contains 2 domain classes, first is a class which represents the

**Money**, second is the **Vendingmachine** itself.

|  |
| --- |
| Money.cs |
| Let’s explain this class step by step. As our machine will only except next cents & coins:   * 5 eurocent,10 eurocent, 20 eurocent, 50 eurocent. * 1 euro-coin, 2 euro-coin.   So I modeled the domain class for money accordingly. Let’s go through the class step by step: |
| **Let’s start with the class header:** |
|  |
| **Next let’s view the public interface of money, containing**  **µall possible values that a money instance can represent:** |
| Explanation: a money is represented by an array of money types (5 in total, varying from 5cent to 2euro).  So first position represents the number of 5cents, second the number of 10cents, and so on,  until the last (6th) which represents the number of 2euro coins. |
|  |
| **We also need the appropriate properties to count the number**  **of money types we inserted into the machine:** |
|  |
| **Next we have the constructor of our money class.**  **This class takes the number of cents and**  **coins per type and creates the money object from it.**  **It also makes sure that no negative money values can be entered.** |
|  |
| **Next I created the “Amount” property which will calculate**  **the value of the money object (by counting the number of**  **cents/coin multiplying by their value).** |
|  |
| **Finally I have added 2 methods to the money class, 1 to sum to moneys (+) operator and 1 to subtract 2 moneys (-) operator.** |
|  |
| **Just to get an idea how to use the money object, some actions we can execute upon them and their result:** |
|  |

|  |
| --- |
| VendingMachine.cs |
| In our context, Vendingmachine takes 2 money types, one to represent the money inside the machine (MoneyInside) and on in transaction (MoneyInTransaction). The first represents the sum of all the sales, the latter is the money put in by the last customer. The use-cases of the vending machine are simple, either accept or return money, depending on the case the customer bought a snack or wants his money back … |
|  |

1. Implementing the test scenarios for Money

Next we will implement some basic unit test scenario’s to test the robustness of our Money instance.

To make the use of SpecFlow clear we will do 2 things: first make a technical unit-test only (like we would

do in the traditional way). Second rewrite the test in a SpecFlow way. So let’s start with some tests:

6.1. Test-1 : Sum of 2 moneys produces correct result

|  |
| --- |
| **Traditional** |
| And the result:    So in a “traditional approach”, we create 2 money objects and add a value to them, next we calculate the sum and display it on the screen. Even though this example is quit simple, it already contains a lot of “technical” C# specific implementation code, and we can avoid this by rewriting the test and add a **“SpecFlow-Wrapper”**. This means that the code above could be re-written in a non-technical **business oriented** way like shown below: |
| **Specflow** |
| Ok, so in case of SpecFlow, the test to sum 2 moneys would be: |
| Well, I think this kind of test is already much more meaningful to the reader. It is more verbose, and more important: written in a natural language (English) thus clearly understandable for people who are not skilled in development languages like C#. Of course this is just a functional “wrapper” and starting point for the developer. Next we will have to map this to specific C# test method stubs, but that’s something totally transparent for both product owner and functional / business analyst or domain expert (in terms of DDD). |
| Currently our feature scenario **is only functionality defined** and **has no-technical-mapping** yet. You can see that the **Given,And,When,Then steps are still in purple-color**, which means **they have no technical implementation** yet. So next, we should first create mapping-stubs between the functional feature scenario and the technical implementation we have to realize yet. We can add these initial subs **by right-clicking the appropriate feature scenario**, and select **“Generate Step Definitions”** from the roll-out menu, this will give us a screen a shown above. So the screen above guides us by proposing the parts we should create step definitions for, as shown below: |
|  |
| Next we can hit the **Preview button** and see what will be generated for us, as shown below: |
|  |
| Above is the C# code that will be generated for us. These will **form the “bridge” between our functional definition of the test and the technical implementation** and already guides the development team on how to write their tests and attach the code to the underlying domain model (in our case the Money object). Of course code has to be written yet by the development team, you can see that the **ScenarioContext.Current.Pending()** has to be replaced by the technical testing code. Finally let’s **generate** the **stubs**, this will create a c# file for us, as shown below: |
|  |
|  |
| So at this stage, the C# stubs are provided as implementation **steps.** Next it’s up to the developer to connect the **steps** to the **domain code,** so, in our case this would be equal to: (implementation steps explained step by step) : 😊   |  |  | | --- | --- | |  | ***Important note :***  *Each automated test has normally 4 distinct steps : Initialize, Arrange, Act and Assert. We will go through each of them in sequence !* | |
| **Step1 : Initialize** |
| Using a stack is appropriate here, because we can easily “push” and “pop” instances to/from the stack, and even (what we will see later) use methods like “peek” to look after the sum … |
| **Step 2 : Arrange** |
| This is the “Arrange” part. In our case this method will be called twice, once for each provided money. We push the provided money to our stack (to calculate the sum from later in the act method). |
| **Step 2 : Act** |
| This is the step where the test get’s executed, in our case it’s a “sum” of 2 moneys (which will test the **”+ operator”** of the money class and push the sum result on the stack. |
| **Step 3 : Assert** |
| This is the last step of our test, the assert will check if the calculated amount (which is the sum of 2 moneys) is equal to the expected value (represented by the p0 parameter) |
| **Step 4 : Execution result** |
|  |

6.2. Test-2 : Two money instances are equal if they contain the same money amount

|  |
| --- |
| Scenario |
| Scenario: Two money instances are equal if they contain the same money amount  Given I have *4* twenty cents and *2* ten cents  And my friend has *1* ten cent and *2* twenty cents and *1* fifty cent  Then we both have the same amount of money which is *2.00* euro in total |
| Step definitions |
| [Given(@"I have (.\*) twenty cents and (.\*) ten cents")]  public void GivenIHaveTwentyCentsAndTenCents(int twentyCentCount, int tenCentCount)  {      Money money = new Money(0,tenCentCount,twentyCentCount,0,0,0);      moneys.Push(money);  }    [Given(@"my friend has (.\*) ten cent and (.\*) twenty cents and (.\*) fifty cent")]  public void GivenMyFriendHasTenCentAndTwentyCentsAndFiftyCent(int tenCentCount, int twentyCentCount, int fiftyCentCount)  {      Money money = new Money(0, tenCentCount, twentyCentCount, fiftyCentCount, 0, 0);      moneys.Push(money);  }    [Then(@"we both have the same amount of money which is (.\*) euro in total")]  public void ThenWeBothHaveTheSameAmountOfMoneyWhichIsEuro(decimal expectedAmount)  {      moneys.Push(moneys.Pop() + moneys.Pop());      Money sum = moneys.Peek();      sum.Amount.Should().Be(expectedAmount);    } |
| Result |
|  |

1. Summary

In this document I tried to explain the use of SpecFlow as a bridge between functional and technical testing.

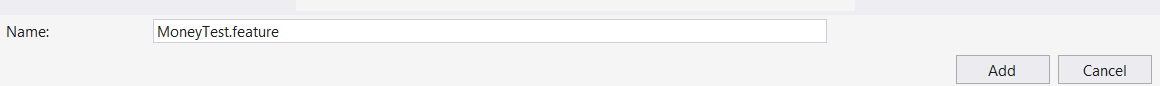
The image below shows a high-level overview of all containing parts, their use and relationships.

|  |
| --- |
|  |
| SpecFlow tests are typically hosted into Unit type of projects. Typically in Visual Studio we would use xUnit but other testing frameworks are valuable to !  SpecFlow uses features and scenario’s (on the functional level) which contain the test-scenario in “natural language, as shown below: |
|  |
| A Feature file contains 1 to many “Scenario’s” and each scenario should test a specific part of the domain model object (for brevity reasons, our test only contains a single scenario, but in real we would have a lot of them, at least enough to cover test all the domain logic. Each scenario will map to it’s technical counterpart, which is represented by a C# step-definitions class, as (partially) shown below: |
|  |
| In fact, **each functional part** which have **it’s technical implementation equivalent**. So a **functional “Given”** will result in **a technical “Given” implementation**, same for the “When” and “Then” steps. This can be clearly deduced from the content above. |

Annex A : Setup feature file

Right-click test project + Add new item …



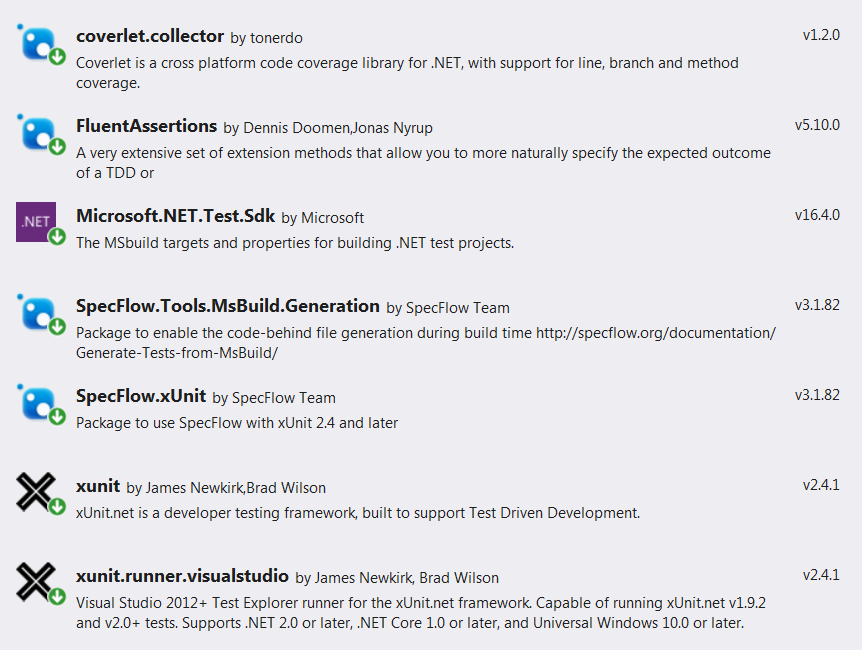


Next we can add some feature scenario. Let’s say we want to test our money object when it takes 2 moneys and

we want to sum them, the correct result is shown on the screen.

Annex B : NuGet package Summary (base: xUnit Testing Project VS)





Annex C : SpecFlow in Depth Course

Introduction

|  |  |
| --- | --- |
| Afbeeldingsresultaat voor goldorak | Goal of this § is to give a more elaborate example on how we can achieve business readable tests using  SpecFlow. As much of the unit-testing originates from the game-world we will implement a game related  Domain model representing a player character from an adventure game which represent the core of the  Domain model for the game, so let us give our domain the generic name of **GameCore.Domain.** The  Main character of our domain is **Goldorak**, a robot from a Japanese cartoon which started in the 70s, I was and still am a big fan of Goldorak ! 😊  on a channel called A2 (Antenne 2). Goldorak has a lot of strengths, more info can be found here:  <https://fr.wikipedia.org/wiki/Goldorak> |
| For the nostalgic fans, some full episodes of “Goldorak” can be found here: 😉  <https://www.youtube.com/watch?v=aOdknJh59aE&list=PLi1QmukUajanTuFSFoK8nc4RWapMaN7k6> | |

Initial Domain class

|  |
| --- |
| As you can see from the code below, our starting domain class is very simple. It contains a (very simplified) representation of a Goldorak Player Character in a adventure game. |
| /// <summary>      /// Author      : Emmanuel Nuyttens      /// Date        : 02-2020      /// Purpose     : Goldorak domain class      /// </summary>      public class Goldorak      {            #region Properties            public int Health { get; protected set; } = 100;          public bool IsDead { get; protected set; } = false;            #endregion Properties            #region Behavior            public void Hit(int damage)          {              Health -= damage;                if(Health <= 0)              {                  IsDead = true;              }          }            #endregion Behavior      } |
| Initial use case:  Currently our Goldorak Character has limited behavior and contains only 2 properties (Health, IsDead) and behavior is limited to the Hit() method. Our initial use case is also very simple:   * When a Goldorak Player Character is created, it has Health set to 100. * Next when it takes “damage” then Health is lowering with Damage. * And if Health reaches 0 then our Player Character is “dead”. |

Creating the first SpecFlow Scenario : Taking no damage when hit doesn’t affect health

|  |
| --- |
| So we will start to create our first **feature file** and add our first **scenario text** within this feature file. Next we will generate our **step-definitions-file**. Finally we will map the functional scenario from our feature file to a technical implementation in our step-definitions-file. Let’s start with having a look at our first scenario: |
|  |
| Feature: Goldorak  *In order to play the Goldorak game*  *As a human player*  *I want my goldorak attributes to be correctly created*    Scenario: Taking no damage when hit doesn't affect health  Given I'm a new Goldorak  When I take 0 damage  Then My health should remain 100 |
| So let’s call our feature “Goldorak” and add a first scenario. The first thing we want to test is that if we create a new instance of our character (Goldorak) and we take no damage, then our character should still possess it’s initial health (which is 100). Ok, next let’s implement this technically by adding a step definitions file: (right-click scenario +Generate Stef definitions), finally this would look like: |
|  |
| You can see that the tool created stubs for our “Given”,”When” and “Then” steps, next save this in the /Stepdefinitions folder (by clicking the “Generate” button). Finally we should get our starting template for our technical implementation as shown below: |
| using System;  using TechTalk.SpecFlow;    namespace GameCore.Specs.StepDefinitions  {      [Binding]      public class GoldorakSteps      {          [Given(@"I'm a new Goldorak")]          public void GivenIMANewGoldorak()          {              ScenarioContext.Current.Pending();          }            [When(@"I take (.\*) damage")]          public void WhenITakeDamage(int p0)          {              ScenarioContext.Current.Pending();          }            [Then(@"My health should remain (.\*)")]          public void ThenMyHealthShouldRemain(int p0)          {              ScenarioContext.Current.Pending();          }      }  } |
| Next we can start writing the test automation code in the provided C# stubs file. |
| [Binding]      public class GoldorakSteps      {          private Goldorak \_goldorak;            [Given(@"I'm a new Goldorak")]          public void GivenIMANewGoldorak()          {              \_goldorak = new Goldorak();          }            [When(@"I take (.\*) damage")]          public void WhenITakeDamage(int damage)          {              \_goldorak.Hit(damage);          }            [Then(@"My health should remain (.\*)")]          public void ThenMyHealthShouldRemain(int expectedHealth)          {              \_goldorak.Health.Should().Be(expectedHealth);          }      } |
| The source code above contains the implementation of our technical step definitions file. If we execute the code we see that the test passes, which means that our domain model is consistent at this moment ! |
|  |
|  |

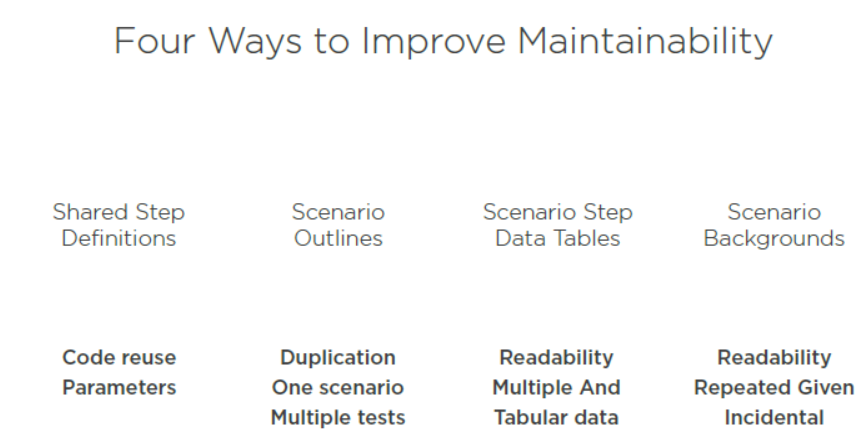
Creating the Second SpecFlow Scenario : Starting Health is reduced when hit

|  |
| --- |
| Let’s first start adding our new scenario to the features file: |
| Scenario: Starting health is reduced when hit  Given I'm a new Goldorak  When I take *40* damage  Then My health should remain *60* |
| As you may note, all steps are already in “black”, which means they are already been bound from the former scenario (because this scenario is similar, it just takes other parameters values for the when and then steps. So, if we ask to create the step-definitions, we will get next message: |
|  |
| And we can just execute the test: |
|  |

Creating the Third SpecFlow Scenario : Taking too much damage results in Goldorak’s death

|  |
| --- |
| Scenario: Taking too much damage results in Goldoraks death  Given I'm a new Goldorak  When I take *100* damage  Then My health should remain *0* |
| As in line with the second scenario, you will see that all steps are already bound, because again, only the provided parameters are different. So if we execute: |
|  |

|  |  |
| --- | --- |
|  | ***Important note :***  *You will have noticed that even though the 3 scenario’s are different, they have a same behavior (meaning reducing health of our character). This means we could reduce this scenario to a single one and provide the parameters as a data-table. We will see how to achieve this later in our demo course.* |



Creating the Fourth SpecFlow Scenario : Adding some businesslogic

|  |
| --- |
| Ok, let’s add some behavior and businesslogic to our Goldorak instance.Let’s assume that the impact of damage that our character takes not only dependson the value of the damage parameter but may be influenced by:The position of Impact (Other,Head,Chest,Legs,Feet).  * The default damage resistance our character has (independent of the position of impact).   So, for our Goldorak figure, we will first add a PostionOfImpact property. We define an enum for this: |
| /// <summary>  /// Author      : Emmanuel Nuyttens  /// Date        : 02-2020  /// Purpose     : Position of damage impact  /// </summary>  public enum PositionOfImpact  {      Other,      Head,      Chest,      Legs,      Feet  } |
| Next we add this property and the notion of default damage resistance to our Goldorak class, as next: |
| public int DefaultDamageResistance { get; set; } = 0;         public PositionOfImpact PositionOfImpact { get; set; } = PositionOfImpact.Other; |
| We also has to adapt our Hit method, so it takes both the default damage resistance and the position of  impact into account when calculating the total amount of damage. So the updated Hit() method is : |
| public void Hit(int damage)  {      var positionOfImpactSpecificDamageResistance = 0;        switch (PositionOfImpact)      {          case PositionOfImpact.Other:              break;          case PositionOfImpact.Head:              positionOfImpactSpecificDamageResistance = 10;              break;          case PositionOfImpact.Chest:              positionOfImpactSpecificDamageResistance = 20;              break;          case PositionOfImpact.Legs:              positionOfImpactSpecificDamageResistance = 30;              break;          case PositionOfImpact.Feet:              positionOfImpactSpecificDamageResistance = 40;              break;          default:              throw new Exception("Unsupported");      };        var totalDamageTaken = Math.Max(damage - positionOfImpactSpecificDamageResistance - DefaultDamageResistance, 0);        Health -= totalDamageTaken;        if(Health <= 0)      {          IsDead = true;      }  } |
| Let’s next add our scenario, so that the new businessrules are taken into account: |
| Scenario: Taken damage depends on damage resistance and position of impact  Given I'm a new Goldorak  And I have a default damage resistance of *20*  And The position of impact is my Head  When I take *40* damage  Then My health should remain *90* |
| And, in our step definitions, we also have to add the necessary technical test implementation code of course : |
| [Given(@"I have a default damage resistance of (.\*)")]          public void GivenIHaveADefaultDamageResistanceOf(int defaultDamageResistance)          {              \_goldorak.DefaultDamageResistance = defaultDamageResistance;          }            [Given(@"The position of impact is my Head")]          public void GivenThePositionOfImpactIsMyHead()          {              \_goldorak.PositionOfImpact = PositionOfImpact.Head;          } |
| And when we test: |
|  |
| The test succeeds. So, while we had an impact of 40 damage, because of our default damage resistance of 20 and head specific damage resistance of 10 (which makes 30), the impact of the damage was only 10, so starting from full health, health remains 90 in this case ! |

Scenario Outlines

Another way to increase maintainability and avoid duplication is achieved by so-called “Scenario-Outlines”,

which will allow us to execute the same basic scenario multiple times, but each time with different test data.

|  |  |
| --- | --- |
|  | ***Info:***  *Using scenario outlines allows the reduction or elimination of repeated scenarios where the only difference between the scenarios are the inputs or expected outcomes. If it feel’s like your scenario is going this direction, then outlines should be used.* |

|  |
| --- |
| Ok let’s rewrite the scenario **“Taken damage depends on damage resistance and position of impact”** to use scenario outlines. So first let’s comment the current implementation of the scenario: |
| #I will rework this scenario so we can have multiple test with different values  #Scenario: Taken damage depends on damage resistance and position of impact  # Given I'm a new Goldorak  # And I have a default damage resistance of 20  # And The position of impact is my Head  # When I take 40 damage  # Then My health should remain 90 |
| And let’s add this one: |
| Scenario Outline: Taken damage depends on damage resistance and position of impact (outline)  Given I'm a new Goldorak  And I have a default damage resistance of *<defaultDamageResistance>*  And The position of impact is *<positionOfImpact>*  When I take *<damageTaken>* damage  Then My health should remain *<expectedHealth>*  Examples:  |*defaultDamageResistance*|*positionOfImpact*|*damageTaken*|*expectedHealth*|  | 0                       | Other            | 10          | 90             |  | 20                      | Head             | 40          | 90             | |
| Ok, I guess the above needs some clarification 😊. So, what I basically did is re-write the scenario to take parameter values instead of fixed values. Next we put the parameter values in an outlined grid. This way we can make a single scenario work for different test use cases. As a first test I added 2 data-rows. Because we changed the behavior of the scenario, we still have to implement a missing scenario step(indicated in purple), which is explained below: |
|  |
| And the implantation : |
| [Given(@"The position of impact is (.\*)")]  public void GivenThePositionOfImpactIs(string positionOfImpact)  {      try        {          \_goldorak.PositionOfImpact = (PositionOfImpact)Enum.Parse(typeof(PositionOfImpact), positionOfImpact);      }      catch (Exception ex)      {          throw ex;      }  } |
| And when we test: |

|  |  |
| --- | --- |
|  | ***Important note :***  *At this point, we parametrize the positionOfImpact as a string and we have to parse it back to the corresponding enum value. Although this is a valid approach, SpecFlow also has the notion of typed objects, so we pas the enum, well … as an enum instead. But we will see later how to do this. Parsing the string to enum is ok for now 😊.* |

Using Data tables in Scenario Steps

In the last scenario (**“Taken damage depends on damage resistance and position of impact”**), we used a data table to execute a whole scenario multiple-times,

but as an alternative we could also use a data table in scenario step(s), so for a specific step or steps within a scenario.

So to test this, we are going to add a new test scenario for our Goldorak character which states:

***“Head PositionOfImpact gets additional 10 default damage resistance when hit”.***

This means that when our character gets hit on the Head, then (next to the already 10 assigned resistance points for HEAD) we get another 10 damage resistance points.

|  |
| --- |
| So before using the data table, let’s implement the scenario in a common way: |
| Scenario: Head PositionOfImpact gets default 10 damage resistance when hit  Given I'm a new Goldorak  And I have a default damage resistance of *10*  And The position of impact is *Head*  When I take *40* damage  Then My health should remain *80* |
| So, all the steps are already bound. This test will pass because starting from a 100-health, setting a default damage resistance of 10 and an additional 10 because our Head got hit, we have 20 in total, so when we take 40 damage, we still have 80 health. |
| What we will do now is that we will replace the multiple “And” statements with a data-table, as next: |
| Scenario: Head PositionOfImpact gets default 10 damage resistance when hit data table  Given I'm a new Goldorak  And I have the following attributes  |*attribute*|*value*|  | PositionOfImpact | Head  |  | Resistance       | 10    |      When I take *40* damage  Then My health should remain *80* |
| What we still have to do is generate the missing definitions step: |
| And implement it: |
| [Given(@"I have the following attributes")]  public void GivenIHaveTheFollowingAttributes(Table table)  {      // get value for first row (PositionOfImpact)      var positionOfImpact = table.Rows.First(row => row["attribute"] == "PositionOfImpact")["value"];        // get value for second row (Resistance)      var resistance = table.Rows.First(row => row["attribute"] == "Resistance")["value"];        try        {          \_goldorak.PositionOfImpact = (PositionOfImpact)Enum.Parse(typeof(PositionOfImpact), positionOfImpact);      }      catch (Exception ex)      {          throw ex;      }        int defaultDamageResistance;        if (Int32.TryParse(resistance, out defaultDamageResistance))      {          \_goldorak.DefaultDamageResistance = defaultDamageResistance;      }      else throw new InvalidCastException();  } |
| What this step definition does is extract the values from the data table (table) and assign to the attributes of our character. Next we can execute the test code, which will pass: |
|  |

|  |  |
| --- | --- |
|  | ***Info:***  *Please note that currently getting the values out of the table is a bit cumbersome but I will provide better solutions later in the course when I will talk about* ***strongly typed table data***      // get value for first row (PositionOfImpact)      var positionOfImpact = table.Rows.First(row => row["attribute"] == "PositionOfImpact")["value"];        // get value for second row (Resistance)      var resistance = table.Rows.First(row => row["attribute"] == "Resistance")["value"]; |

Working with Data in Step Definitions

In the scenario’s before we talked about parameterized step definitions and data table, in this section I’m going to detail a bit more some of these

concepts and learn how SpecFlow converts **text in our scenario steps** into **.NET data types** and how we can take control of this process.

|  |
| --- |
| Let’s first talk a bit about **step argument conversion**, which is taking the **plain text in our feature file** and converting it to a **.NET data type in our step definition parameter.** Let’s take next when step to illustrate:  When I take *40* damage  So in this example we have the text “40”. In our step definition we create parameters that match pieces of the text in our scenario. So in this case we could have a **when step** in our functional feature file and **an attribute in our** technical step definition C# source file to map to, as shown below:  [When(@"I take (.\*) damage")]  public void WhenITakeDamage(int damage)  This value that we’ve matched in our scenario text is still just plain text. In our step definition we have this parameter called **damage** of type **int.** So in our case the **text “40” of our functional scenario** will be translated to an **int type of 40 in our technical step definitions** source file. This will be transparent for the user ! (auto-conversion takes place here). |
| It is important to know that there are a different number of ways in which the text from our scenario’s gets converted to a .NET data type, and there is an important order of precedence here as well ! :   * **First in the order of precedence: No Conversion Necessary.**   🡪So if the type of our parameter in our method is either object or string, then auto-conversion is done.   * **Second in the order of precedence: Custom Step Argument Transformation.**   🡪This type of conversion will happen when we defined a custom transformation that matches the parameter type in our step definition.   * **Last in the order of precedence: In case Conversion is required but no Matching Custom Transformation is registered.**   🡪 Then any standard or inbuilt conversions will take place by SpecFlow. So for example if the text value can be converted to the .NET data type using the Convert.ChangeType() method then SpecFlow will do this for us ! And if the type of our parameter is an enum type, then SpecFlow will attempt to convert the text to one of the enum values. Finally, if the type of the parameter is a GUID, once again, SpecFlow will attempt to convert the provided text to a GUID. |

|  |
| --- |
| So to illustrate this, let’s first add some behavior to our Goldorak Character domain class. We will make our Goldorak Character a bit less vulnerable by adding a new **MagicalItem class** and new **Weapon class,** a new **Goldorak class enumeration** and new **Goldorak methods/properties.** Pfjew … 😉 |
| **MagicalItem** |
| |  |  |  |  | | --- | --- | --- | --- | | Name | Image | Value | C# Class | | FulguroPoint | C:\Users\emmnuy\AppData\Local\Microsoft\Windows\Temporary Internet Files\Content.MSO\6905964.tmp | 5 | /// <summary>  /// Author      : Emmanuel Nuyttens  /// Date        : 02-2020  /// Purpose     : Magical Item class  /// Info        : Represents an item of magical power in the  ///               Goldorak game such as:  ///               -fulguro point, asterohache, retrolaser, pulvonium, cornofulgure  ///               -planitron, clavicogyre, megavolts, missilesgamma  ///  /// </summary>  public class MagicalItem  {      public string Name { get; set; }      public int Value { get; set; }      public int Power { get; set; }    } | | AsteroHache | https://dessins-animes-hrd.appspot.com/img/dessins-animes-2/Goldorak/goldorak-asterohache2.jpg | 10 | | RetroLaser | C:\Users\emmnuy\AppData\Local\Microsoft\Windows\Temporary Internet Files\Content.MSO\D88D754E.tmp | 15 | | PulVonium | C:\Users\emmnuy\AppData\Local\Microsoft\Windows\Temporary Internet Files\Content.MSO\4371A07A.tmp | 20 | | CornoFulgure | C:\Users\emmnuy\AppData\Local\Microsoft\Windows\Temporary Internet Files\Content.MSO\6C782E98.tmp | 25 | | PlaniTron | C:\Users\emmnuy\AppData\Local\Microsoft\Windows\Temporary Internet Files\Content.MSO\76527E7E.tmp | 30 | | ClavicoGyre | C:\Users\emmnuy\AppData\Local\Microsoft\Windows\Temporary Internet Files\Content.MSO\BF645912.tmp | 35 | | MegaVolts | C:\Users\emmnuy\AppData\Local\Microsoft\Windows\Temporary Internet Files\Content.MSO\834AAAF0.tmp | 40 | | MissilesGamma | C:\Users\emmnuy\AppData\Local\Microsoft\Windows\Temporary Internet Files\Content.MSO\A56EC866.tmp | 45 | |
| **TeamMate** |
| |  |  |  |  | | --- | --- | --- | --- | | Name | Image | Value | C# Class | | OvTerre | OVT1.jpg | 10 | /// <summary>      /// Author      : Emmanuel Nuyttens      /// Date        : 02-2020      /// Purpose     : Team mates of Goldorak      /// Info        : -OvTerre (Alcor), Venusiac (Venusia), Phosoirac (Phenicia), Aquarak (Venusia)      /// </summary>      public class TeamMate      {          public string Name { get; set; }          public int Value { get; set; }      } | | Venusiac | http://aperoboy.free.fr/images/actarus/images-pics/venusia-goldorak.jpg | 20 | | Phosoirac | http://aperoboy.free.fr/images/actarus/images-pics/phen-fossouarac.jpg | 30 | | Aquarak | http://aperoboy.free.fr/images/actarus/images-pics/phenicia-vaisseau.jpg | 40 | |
| **UfoState** |
| |  |  |  |  | | --- | --- | --- | --- | | Enum | Image | Value | C# Enum | | Docked | C:\Users\emmnuy\AppData\Local\Microsoft\Windows\Temporary Internet Files\Content.MSO\6F84B248.tmp | 0 | /// <summary>      /// Author      : Emmanuel Nuyttens      /// Date        : 06-2020      /// Info        : State of Goldorak      ///               either docked or undocked      /// </summary>      public enum UfoState      {          Docked,          Undocked      } | | Undocked | C:\Users\emmnuy\AppData\Local\Microsoft\Windows\Temporary Internet Files\Content.MSO\C684F796.tmp | 1 | |  |  |  | |  |  |  | |
| **Goldorak (Updates)** |
| Next we also have some modifications in our Goldorak class, let’s start with **MagicalItems** and **MagicalPower** |
| public int MagicalPower  {      get { return MagicalItems.Sum(magicalItem => magicalItem.Power); }  }    public List<MagicalItem> MagicalItems { get; set; } = new List<MagicalItem>();      public int TeamMatesValue  {      get { return TeamMates.Sum(teamMate => teamMate.Value); }  }    public List<TeamMate> TeamMates { get; set; } = new List<TeamMate>();  public UfoState UfoState { get; set; } = UfoState.Attached; |
| Our character holds a list of magical items, this is represented by a list of **MagicalItem** properties and we also keep track of the total power of the magical items, this is represented by **MagicalPower** property. When entering a battle, our Goldorak figure may also be accompanied by some team mates, this is represented by a list of **TeamMate** and total value of all teammates, represented by the **TeamMatesValue** property and also Goldorak may be or not be “attached” to his “soucoupe”, this is represented by it’s **“UfoState”**. |
| public DateTime LastMaintenanceDate { get; set; } |
| Like all Ufo’s, our Golodrak character has to be maintained, so we keep track of it’s last maintenance date. |
| public void RepairHealth()  {      if(UfoState == UfoState.Docked)      {          Health = 100;      }      else      {          Health = Health < 100 ? Health + 10 : Health;      }  } |
| I also added a method to repair the health of Goldorak. The business rule states that when Goldorak is “Docked” then het get’s back 100, else Health is +10, and of course Health can not pass 100… |
| public void ReadHealthScroll()  {      var daysSinceMaintenance = DateTime.Now.Subtract(LastMaintenanceDate).Days;        if(Health < 100 && daysSinceMaintenance <= 2)      {          Health = 100;      }  } |
| Next I added a method that checks the maintenance date, and the business rule here states that when our Goldorak has got maintenance within the last 2 days, he get’s all health back ! |
| public void UseMagicalItem(string itemName)  {      try      {            int powerReduction = 10;            if (UfoState == UfoState.Docked)          {              powerReduction = 0;          }            var itemToReduce = MagicalItems.First(item => item.Name == itemName);            itemToReduce.Power -= powerReduction;            itemToReduce.Power = itemToReduce.Power < 0 ? 0 : itemToReduce.Power;      }      catch (Exception ex)      {          throw ex;      }  } |
| Next I also added a method to use a magical item. And a business rule states that only when Goldorak is docked, then no power reducation is applied. |

Automatic Enum Conversion

|  |
| --- |
| \_goldorak.PositionOfImpact = (PositionOfImpact)Enum.Parse(typeof(PositionOfImpact), positionOfImpact); |
| As shown above, former implementation of an enum conversion was a bit cumbersome, we had to do a lot of typecasting before we could get the enum out. I will create a new scenario to show how we can do this in a more clean way. |
| **Scenario: “UfoState Docked restores all health”.** |
| Let’s start by adding a new feature in our functional feature file:  Scenario: UfoState Docked restores all health  Given I'm a new Goldorak  Given My Goldorak character ufo state is Docked  When I take *40* damage  And  Execute a repair health request  Then My health should remain *100* |
| So the test scenario above will create a new character, set the ufo state to docked and take 40 damage. Next when repair health request is executed, health should be back to 100. You can see that some step definitions should be generated yet (the purple lines), so let’s do that first: |
|  |
| Copy to clipboard …and add them to our technical step definitions file, as next: |
| [Given(@"My Goldorak character ufo state is Docked")]  public void GivenMyGoldorakCharacterUfoStateIsDocked()  {      ScenarioContext.Current.Pending();  }    [When(@"Execute a repair health request")]  public void WhenExecuteARepairHealthRequest()  {      ScenarioContext.Current.Pending();  } |
| And implement them … |
| [Given(@"My Goldorak character ufo state is (.\*)")]  public void GivenMyGoldorakCharacterUfoStateIsDocked(UfoState ufoState)  {      \_goldorak.UfoState = ufoState;  }    [When(@"Execute a repair health request")]  public void WhenExecuteARepairHealthRequest()  {      \_goldorak.RepairHealth();  } |
|  |
| You can see that the value of the text “Docker” is correctly mapped to the corresponding value of the enum. |
|  |
| And our test passes 😊 |

Strongly-typed Step Table Data

Let’s have a look how we can change the scenario that uses the data-table to use strongly-typed data instead.

|  |
| --- |
| Scenario: Head PositionOfImpact gets default 10 damage resistance when hit data table  Given I'm a new Goldorak  And I have the following attributes  |*attribute*|*value*|  | PositionOfImpact | Head  |  | Resistance       | 10    |      When I take *40* damage  Then My health should remain *80* |
|  |
| First we go to the step-definition for: **“And I have the following attributes”** as shown above. Instead of accessing the data in the rows in a “weakly-typed” manner, we are going to create a new class to represent these data items. We will add this class in our Specs test project (GameCore.Specs):  /// <summary>  /// Author      : Emmanuel Nuyttens  /// Date        : 02-2020  /// Purpose     : Attributes class for strongly-typed table-data  /// </summary>  public class GoldorakAttributes  {      public PositionOfImpact PositionOfImpact { get; set; }      public int Resistance { get; set; }  } |
| Next we go to our step definition file and first add a new using statement on top: |
| using TechTalk.SpecFlow.Assist; |
| Finally we can adapt our step definition method ax next: |
| [Given(@"I have the following attributes")]  [Given(@"I have the following attributes")]  public void GivenIHaveTheFollowingAttributes(Table table)  {        var attributes = table.CreateInstance<GoldorakAttributes>();      \_goldorak.PositionOfImpact = attributes.PositionOfImpact;      \_goldorak.DefaultDamageResistance = attributes.Resistance;    } |
| The former code has been commented out. And you can see that using the new approach our code is much easier to understand. |

Dynamic Step Data Table

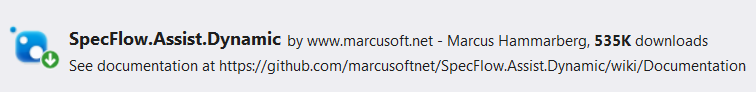
This is an alternative to the former scenario, and here we will not need to add the extra attribute class (in our case: GoldorakAttributes). So we will

Again rewrite next scenario step:

**“And I have the following attributes”**

Instead of creating an attribute class, we will take advantage of the **C# dynamic features.**

To be able to use the C# dynamic features, we first have to add a new NuGet package in our test project:



|  |
| --- |
| After installation of the NuGet, we must open our definition step again and change to: |
| [Given(@"I have the following attributes")]          public void GivenIHaveTheFollowingAttributes(Table table)          {              //V3: Dynamic attributes              dynamic attributes = table.CreateDynamicInstance();              \_goldorak.PositionOfImpact = (PositionOfImpact)Enum.Parse(ty  peof(PositionOfImpact),attributes.PositionOfImpact);              \_goldorak.DefaultDamageResistance = attributes.Resistance;            } |
| Advantage of this approach is that we do not need the creation of the attributes class, but we still have to remap our enum ! |

Multi-column Step Table Data

In addition to pass simple key/value style data to step definitions, we can also pass multi-column data: for example a list of complex items. To illustrate this I will add a new scenario called **“Total Magical Power”** that will test the sum of current magical power items a given Goldorak instance possesses.

|  |
| --- |
| In our feature file scenario we will pass a table of magical items to the step definition C# source file. |
| #multi column step table-data  Scenario: Total magical power  Given I'm a new Goldorak  Given I have the following magical items  |*item*|*value*|*power*|  | FulguroPoint  | 5     | 100   |  | AsteroHache   | 10    | 125   |  | RetroLaser    | 15    | 150   |  | Pulvonium     | 20    | 175   |  | CornoFulgure  | 25    | 200   |  | PlaniTron     | 30    | 225   |  | ClavicoGyre   | 35    | 250   |  | MegaVolts     | 40    | 275   |  | MissilesGamma | 45    | 300   |  Then My total magical power should be 1800 |
| Next we still have to generate some missing step defitions: |
|  |
| [Given(@"I have the following magical items")]  public void GivenIHaveTheFollowingMagicalItems(Table table)  {      ScenarioContext.Current.Pending();  }    [Then(@"My total magical power should be (.\*)")]  public void ThenMyTotalMagicalPowerShouldBe(int p0)  {      ScenarioContext.Current.Pending();  } |
| And add them to the step definition file as mentioned above. Finally we have to implement them. As with the example before, I will provide 3 solutions, starting with the **weakly typed** version: |
| [Given(@"I have the following magical items")]  public void GivenIHaveTheFollowingMagicalItems(Table table)  {      //V1 : Weakly typed      foreach(var row in table.Rows)      {          var name = row["item"];          var value = row["value"];          var power = row["power"];            \_goldorak.MagicalItems.Add(new MagicalItem          {              Name = name,              Value = int.Parse(value),              Power = int.Parse(power)          });      }  }  [Then(@"My total magical power should be (.\*)")]  public void ThenMyTotalMagicalPowerShouldBe(int expectedPower)  {      \_goldorak.MagicalPower.Should().Be(expectedPower);  } |
| Second implementation is the strongly typed version: |
| [Given(@"I have the following magical items")]  public void GivenIHaveTheFollowingMagicalItems(Table table)  {      //V2 : Strongly typed      var items = table.CreateSet<MagicalItem>();      \_goldorak.MagicalItems.AddRange(items);  } |
|  |
| But we will currently have a problem, because our MagicalItem class expect a “name” field, not an “item” as provided by the data-table. So we will change the ‘item’ to ‘name’ in the provided data-table of the feature file: |
| #multi column step table-data  Scenario: Total magical power  Given I'm a new Goldorak  Given I have the following magical items  |*name*|*value*|*power*|  | FulguroPoint  | 5     | 100   |  | AsteroHache   | 10    | 125   |  | RetroLaser    | 15    | 150   |  | Pulvonium     | 20    | 175   |  | CornoFulgure  | 25    | 200   |  | PlaniTron     | 30    | 225   |  | ClavicoGyre   | 35    | 250   |  | MegaVolts     | 40    | 275   |  | MissilesGamma | 45    | 300   |  Then My total magical power should be *1800* |
| And now it should be fine: |
|  |
| And the test passes: |
|  |
| And finally we can also use dynamics: |
| [Given(@"I have the following magical items")]  public void GivenIHaveTheFollowingMagicalItems(Table table)  {          //V3 : Dynamic attributes      var items = table.CreateDynamicSet();      foreach(var magicalItem in items)      {          \_goldorak.MagicalItems.Add(new MagicalItem          {              Name = magicalItem.name,              Value = magicalItem.value,              Power = magicalItem.power          });      }    }  Be aware you don’t have intellisense support for name,value,power when using dynamics. |

Custom Data Conversions

One of the powerful features of SpecFlow is our ability to provide **custom data conversions**. I will illustrate this use by providing a new scenario:

“Requesting a restore health when not maintained properly has no effect”.

So this basically means that when our Goldorak character is not properly maintained, requesting a restore health will have not effect. Let’s implement this:

|  |
| --- |
| Let’s start by adding our new scenario: |
| Scenario: Requesting a restore health when not maintained properly has no effect  Given I'm a new Goldorak  Given I last had maintenance 3 days ago  When I take *40* damage  And I request as restore health  Then My health should remain *60* |
| Let’s generate the missing step definition (in purple): |
|  |
| And add them to our step-definitions file: |
| [Given(@"I last had maintenance (.\*) days ago")]  public void GivenILastHadMaintenanceDaysAgo(int p0)  {      ScenarioContext.Current.Pending();  }    [When(@"I request as restore health")]  public void WhenIRequestAsRestoreHealth()  {      ScenarioContext.Current.Pending();  } |
| And finally implement them: |
| [Then(@"My total magical power should be (.\*)")]  public void ThenMyTotalMagicalPowerShouldBe(int expectedPower)  {      \_goldorak.MagicalPower.Should().Be(expectedPower);  }    [Given(@"I last had maintenance (.\* days ago)")]  public void GivenILastHadMaintenanceDaysAgo(DateTime lastMaintenanceDate)  {      \_goldorak.LastMaintenanceDate = lastMaintenanceDate;  }    [When(@"I request as restore health")]  public void WhenIRequestAsRestoreHealth()  {      \_goldorak.ReadHealthScroll();  } |
| What we still have to do now is convert the text parameter from the “Given” attribute into a Date Time type of parameter, that’s where the custom data conversions class comes in. We will create a new class for this in our Spec Test project: |
| /// <summary>  /// Author      : Emmanuel Nuyttens  /// Date        : 02-2020  /// Purpose     : Custom conversions class  /// </summary>  [Binding]  public class CustomConversions  {      [StepArgumentTransformation(@"(\d+) days ago")]      public DateTime DaysAgoTransformation(int daysAgo)      {          return DateTime.Now.Subtract(TimeSpan.FromDays(daysAgo));      }  } |
| So in our case, we are using regular expressions to convert the text into a number of days ago integer variable. Next we use this int variable to calculate and return the last maintenance date. So if we execute & debug: |
| We see that our method extracts the number of days and the test passes: |
|  |

Automatically Applying Custom Transforms

Let’s next have a look how we can automatically apply transforms when we have a certain type of data. To illustrate this I will create a new scenario:

**“Teammates are worth value”.**

This means, that, if we have a number of **team mates**, then the total value of these team mates is correct. So let’s start with the implementation of this scenario:

|  |
| --- |
| We start by adding our scenario in our feature file: |
| Scenario: Teammates are worth value  Given I'm a new Goldorak  Given I have the following team mates  |*name*|*value*|  | ovterre   | 10    |  | venusiac  | 20    |  | phosoirac | 30    |  | aquarak   | 40    |  Then my team mates should be worth 100 |
| So the test is quit simple, we create a new instance of our Goldorak, add some team mates to them and then test the sum of the value of the team mates. As always, let first create the missing step definitions: |
|  |
| And add them to the step definitions file: |
| [Given(@"I have the following team mates")]  public void GivenIHaveTheFollowingTeamMates(Table table)  {      ScenarioContext.Current.Pending();  }    [Then(@"my team mates should be worth (.\*)")]  public void ThenMyTeamMatesShouldBeWorth(int p0)  {      ScenarioContext.Current.Pending();  } |
| Next let’s first implement the “then” step, which check the total amount of team mates value: |
| [Then(@"my team mates should be worth (.\*)")]  public void ThenMyTeamMatesShouldBeWorth(int value)  {      \_goldorak.TeamMatesValue.Should().Be(value);  } |
| Next let’s implement the ‘Given’ step. Here we will have a list of team mates and their values and we want to add them to Goldorak’s list of team mates. We will first start with replacing the ‘table’ parameter by an IEnumerable of Team Mates. |
| [Given(@"I have the following team mates")]  public void GivenIHaveTheFollowingTeamMates(IEnumerable<TeamMate> teamMates)  {      \_goldorak.TeamMates.AddRange(teamMates);  } |
| Because SpecFlow won’t currently be able to do the transformation from ‘table’ to ‘IEnumerable<TeamMate>’ we will have to instruct SpecFlow how to do this by providing a ‘custom conversion’ (like we did already before). So in our already existing ‘CustomerConversion’ class, we have to add this: |
| [StepArgumentTransformation]  public IEnumerable<TeamMate> TeamMatesTransformation(Table table)  {      var teamMates = table.CreateSet<TeamMate>();        return teamMates;  } |
| Ok, let’s debug & test : |
|  |
| As you can see from above images, yhe values of our teammates are well transformed by the conversion class and supplied to our step definitions source file ! 😊 And the test succeeds : |
|  |

Passing Data Between Step Definitions

It can sometimes be necessary to pass data between different step definitions of a given scenario. This means storing the data in one step and retrieving this data in the other. With SpecFlow, we can accomplish this in either one of these 3 ways:

1. Step class fields or properties
   * E.g. passing instance of a new goldorak between different steps.
2. SpecFlow provided context objects.
   * Here we can store data in context and pass them between step definitions.
   * We also have tread safe versions to be used when tasks are executed in parallel.
   * Context sharing is using weakly-typed dictionary type of objects <key,value> pair, so we still have to cast to the right object type.
3. Custom context object injection.
   * Here we have to write some additional code in custom classes.
   * Because we write our own custom objects, these can be handled as strongly-typed, so we avoid casting here.
   * Is thread safe.

In practice, we will be able to share data on 3 levels: feature, scenario and scenario-steps. For each there is a context provided, and this context can contain data that can be shared on the different levels, as shown in the image below:

|  |  |
| --- | --- |
|  | So in the example shown on the left, we have a shared context for our whole feature file, next a shared context for each scenario and next a shared context for each of the scenario steps within a scenario.    All 3 of the context types provide a weakly typed **Dictionary<string,object>.** |
| **Example Scenario Context (Not Thread Safe 🡪 so don’t use in case of parallel tasks)** | |
| //# passing data : ScenarioContext (not thread safe)  [Given(@"I have a magic item with a power of (.\*)")]  public void GivenIHaveAMagicItemWithAPowerOf(int power)  {      ScenarioContext.Current["power"] = power;  }  [Then(@"The magic item power should not be reduced")]  public void ThenTheMagicvItemPowerShouldNotBeReduced()  {      int expectedPower = (int)ScenarioContext.Current["power"];  } | |
| In the case written above we will store the value of **power** argument of the magic item in the **scenario context**. We can then retrieve this value later on in the **Then** step by accessing the static **Current** property of the **ScenarioContext class**, and once again, using the **power** as the **key** here. | |
| **Example Scenario Context (Thread Safe 🡪 use this one in case of parallel tasks)** | |
| //# passing data : ScenarioContext (thread safe)  [Given(@"I have a magic item with a power of (.\*)")]  public void GivenIHaveAMagicItemWithAPowerOf(int power)  {      this.ScenarioContext.Current["power"] = power;  }  [Then(@"The magic item power should not be reduced")]  public void ThenTheMagicvItemPowerShouldNotBeReduced()  {      int expectedPower = (int)this.ScenarioContext.Current["power"];  } | |
| Same code as before, but we are using **this**.ScenarioContexton our binding class. Later on in the **Then** step we are also retrieving the **power** from the **ScenarioContext**, again from thisinstance of our binding class. But to make this possible, our step definition class should inherit from next: | |
| [Binding]  public class GoldorakSteps : TechTalk.SpecFlow.Steps  { | |
| This way we have access to our context in a thread safe way. | |

Using Context Injection

Context injection enables us to share data between steps in a thread safe and strongly typed way. I will show how we can use

Context injection to pass strongly typed data between definition steps. To illustrate this I will, again, create a new scenario called:

**“Goldorak does not lose magical power when docked”**

This is a special feature of Goldorak that adds a business rule which says if Goldorak uses magical power while docked, the magical

power of these magical items doesn’t get reduced when Goldorak is docked to his station. To know the difference between the initial

start power and the power after Goldorak uses it’s magical power item, we have to cache the original starting power value of used

magical item. That’s where the context injection comes in place. First we will have to create a context class where we can add the

necessary items that should be cached (in our case, this will be the starting power of the used magical power), so our class will

look like next:

/// <summary>

/// Author      : Emmanuel Nuyttens

/// Date        : 02-2020

/// Purpose     : Share context data between step definitions

/// </summary>

public class GoldorakStepsContext

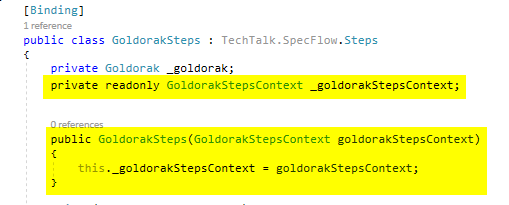
{

    public int StartingMagicalPower { get; set; }

}

And we will need to inject an instance of this class in our step definition code, so we can use it to cache the starting power

of the supplied magical item for our test:



So after adding the context initialization, let’s implement this in our scenario:

|  |
| --- |
| So let’s start by adding our scenario which states if goldorak is docked and uses a magical item (like MegaVolts) in case he’s UfoStatus is docked, then this magical power won’t be reduced: |
| Scenario: Goldorak does not lose magical power when docked  Given I'm a new Goldorak  And I'm Docked to my base station  And I have a magical item MegaVolts with a power of 275  When I use a magical item MegaVolts  Then The magical item MegaVolts power should not be reduced |
| Next let’s create the missing definitions first: |
|  |
| And add them to our step definitions file, as next: |
| [Given(@"I'm Docked to my base station")]  public void GivenIMDockedToMyBaseStation()  {      ScenarioContext.Current.Pending();  }    [Given(@"I have a magical item MegaVolts with a power of (.\*)")]  public void GivenIHaveAMagicalItemMegaVoltsWithAPowerOf(int p0)  {      ScenarioContext.Current.Pending();  }    [When(@"I use a magical item MegaVolts")]  public void WhenIUseAMagicalItemMegaVolts()  {      ScenarioContext.Current.Pending();  }    [Then(@"The magical item MegaVolts power should not be reduced")]  public void ThenTheMagicalItemMegaVoltsPowerShouldNotBeReduced()  {      ScenarioContext.Current.Pending();  } |
| And finally let’s implement them one by one : |
| [Given(@"I'm Docked to my base station")]  public void GivenIMDockedToMyBaseStation()  {      \_goldorak.UfoState = UfoState.Docked;  } |
| We have a given step to implement that sets Goldorak UfoStatus to docked. |
| public void GivenIHaveAMagicalItemMegaVoltsWithAPowerOf(int power)  {      // create a magical item of megavolts      \_goldorak.MagicalItems.Add(new MagicalItem      {          Name = "MegaVolts",          Power = power      });        // add starting power to the shared context      // so we can use it in other step definitions      \_goldorakStepsContext.StartingMagicalPower = \_goldorak.MagicalItems[0].Power;  } |
| This is the interesting part where in a certain step will have to store shared data. In our case we have to store the initial power of the added magical item, this is because this value may be affected by our business rules in later steps. |
| [When(@"I use a magical item MegaVolts")]  public void WhenIUseAMagicalItemMegaVolts()  {      \_goldorak.UseMagicalItem("MegaVolts");  } |
| In this step the business logic is executed, in our case this will be our Goldorak which is using it’s magical item “Megavolts”. This action may, depending on it’s UfoStatus reduce or not reduce the power of the used magical item. |
| [Then(@"The magical item MegaVolts power should not be reduced")]  public void ThenTheMagicalItemMegaVoltsPowerShouldNotBeReduced()  {      int expectedPower = \_goldorakStepsContext.StartingMagicalPower;        \_goldorak.MagicalPower.Should().Be(expectedPower);  } |
| Finally we come into the then method of our test, where the assert is executed. In our case we will check if the expected power (starting power which was cached by the context) is equal to the current power of the MagicalPower. In our case, this test will pass: |
|  |

Controlling Test Execution and Running Additional Code

Introduction

In this section I will explain how we can control the execution of our SpecFlow tests and also how we can run some

additional automation code behind the scenes at various points during the test execution lifecycle.

Next we will learn about the parallel execution features of SpecFlow, and how this can reduce the time that our tests

take to execute.

Using Tags to Execute Subsets of Tests

Let’s have a look how we could use “tags” (@<tag-name>) to execute subsets of our tests. So to illustrate the use of tags for our

Goldorak character, we will add a (UfoStatus).Docked tag to specific scenarios that deal with Docked-specific attributes

(this means attributes in form of magical items that only can be used when Goldorak is Docked to it’s base station).

|  |
| --- |
| So first we will tag all our scenario’s which have something to do with “Docked” UfoStatus: |
| *@docked*  Scenario: UfoState Docked restores all health  Given I'm a new Goldorak  Given My Goldorak character ufo state is *Docked*  When I take *40* damage  And  Execute a repair health request  Then My health should remain *100*  *@docked*  Scenario: Goldorak does not lose magical power when docked  Given I'm a new Goldorak  And I'm Docked to my base station  And I have a magical item MegaVolts with a power of *275*  When I use a magical item MegaVolts  Then The magical item MegaVolts power should not be reduced |
| So next, after we re-build the solution and open the test-explorer in Visual Studio, we can group items by Traits (which is equivalent to tags in SpecFlow): |
|  |
| |  |  | | --- | --- | |  |  |   We could decide to only show the scenario tagged with “docked” & execute them … |
| We can also use the @ignore tag, which will ignore the specific scenario: |
| *@docked @ignore*  Scenario: Goldorak does not lose magical power when docked |
| And when we execute: |
|  |
| So the specific scenario has been ignored to execute. |

Restricting Step Execution with Scoped Bindings

In the case we have multiple step definitions that have the same bindings, but we want to differentiate which step

definition gets executed, then we can do this with scoped bindings. So in this demo we are going to create a different

step definition that matches the same **“I take damage”,** but this time we only want this step definition to execute if the

scenario has been tagged as belonging to “Docked” Goldorak characters.

|  |
| --- |
| So to illustrate this, we will have 2 scenario’s now, the first is still the default one, but the second one explicitly sets the UfoState to Docked, set maintenance to today and calls a ReadHealthScroll which repairs the health, the other one is still the default one which will not repair health (and either doesn’t set the UfoState explicitly, that’s a bit of a quirk I know, it should set UfoState to UnDocked but I did not want to do this to not possibly influence other tests, I just wanted to illustrate the combination of tag/scope). |
| Scenario: Starting health is reduced when hit undocked  Given I'm a new Goldorak  When I take *40* damage  Then My health should remain *60*  Scenario: Starting health is repaired when hit docked  Given I'm a new Goldorak  When I take *40* damage  Then My health should remain *100* |
| [When(@"I take (.\*) damage")]  public void WhenITakeDamage(int damage)  {      \_goldorak.Hit(damage);  }    [When(@"I take (.\*) damage")]  public void WhenITakeDamageAsDocked(int damage)  {      \_goldorak.UfoState = UfoState.Docked;      \_goldorak.Hit(damage);      \_goldorak.LastMaintenanceDate = DateTime.Now;      \_goldorak.ReadHealthScroll();  } |
| Setup of our scenario has changed a bit, instead of assigning the UfoStatus (either docked or undocked) in a given step, we want to differentiate by using a scoped binding. If we execute the scenario’s as mentioned above, we will currently run into an error. |
| |  |  | | --- | --- | |  |  |   We will get an error that we have **Ambiguous step definitions.** So what we have to do is create a tag (either for docked or undocked) in the feature file and apply that tag with a **scoped** attribute in the step definitions file: |
| Scenario: Starting health is reduced when hit undocked  Given I'm a new Goldorak  When I take *40* damage  Then My health should remain *60*    *@docked*  Scenario: Starting health is repaired when hit docked  Given I'm a new Goldorak  When I take *40* damage  Then My health should remain *100*  [When(@"I take (.\*) damage")]  public void WhenITakeDamage(int damage)  {      \_goldorak.Hit(damage);  }    [When(@"I take (.\*) damage")]  [Scope(Tag="docked")]  public void WhenITakeDamageAsDocked(int damage)  {      \_goldorak.UfoState = UfoState.Docked;      \_goldorak.Hit(damage);      \_goldorak.LastMaintenanceDate = DateTime.Now;      \_goldorak.ReadHealthScroll();  } |
| Now the methods will be executed accordingly and the test will pass. |
|  |

Overview of Hooks for Execution additional Code

SpecFlow gives us the functionality to be able to run additional automation code during the test execution lifecycle.

|  |  |
| --- | --- |
|  | We typically have a feature, and within the feature we have at least a scenario. This scenario then contain a number of steps: (Given, And …) , (When) and (Then, And …). This Given and And step form the “Given phase” of the scenario. The single When step forms the When block of the scenario. Finally, the Then and And steps form the Then block.  There are a number of places (hooks) where we can execute additional automation code during the test execution lifecycle. These hooks are implemented as attributes in SpecFlow. |
|  | |
| Executing Code Before and After Every Scenario | |
| Let’s have al look how we can use the **“Before” and “After” scenario attribute** to execute additional automation code before and after each scenario executes. We will first start to add a new class to our spec test project which will hold the “before” and “after” code to execute. We first have to “tag” a scenario which we want to add before/after code hooks for, let’s take the “Starting health is reduced when hit undocked” scenario of the Goldorak-ScopedBindings.Feature and tag it with @undocked: | |
| *@undocked*  Scenario: Starting health is reduced when hit undocked  Given I'm a new Goldorak  When I take *40* damage  Then My health should remain *60* | |
| Next add a new class to hold the before/after code hooks (just for illustration, we won’t add real code though …): | |
| /// <summary>  /// Author  : Emmanuel Nuyttens  /// Date    : 02-2012  /// Purpose : Additional code hooks class  /// Info    : Inherit from Steps base class which gives access to  ///           a thread safe ScenarioInfo  /// </summary>  [Binding]  public class AdditionalCodeHooks : Steps  {      [BeforeScenario("undocked")]      public void BeforeScenario()      {          //TODO: Add code which should be executed before scenario start      }        [AfterScenario("undocked")]      public void AfterScenario()      {          //TODO: Add code which should be executed after scenario ends      }    } | |
| So finally let’s start all of our tests for Goldorak-ScopedBindings.Feature. The hooks get executed appropriately just before and after the scenario steps. | |
|  | |

1. Product Owner [↑](#footnote-ref-1)